# Lightweight verification of control flow policies on Java bytecode

Arnaud Fontaine, Samuel Hym, Isabelle Simplot-Ryl

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Lightweight verification of control flow policies on Java bytecode*

Arnaud Fontaine — Samuel Hym — Isabelle Simplot-Ryl

**N° 7584**

avril 2011

*Rapport de recherche*

ISSN 0249-6399 ISRN INRIA/RR--7584--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
LILLE - NORD EUROPE

# Lightweight verification of control flow policies on Java bytecode

Arnaud Fontaine, Samuel Hym, Isabelle Simplot-Ryl

Theme :
Équipe-Projet POPS

Rapport de recherche n° 7584 — avril 2011 — 22 pages

**Abstract:** This paper presents the enforcement of control flow policies for Java bytecode devoted to open and constrained devices. On-device enforcement of security policies mostly relies on run-time monitoring or inline checking code, which is not appropriate for strongly constrained devices such as mobile phones and smart-cards. We present a *proof-carrying code* approach with on-device *lightweight verification* of control flow policies statically at loading-time. Our approach is suitable for evolving, open and constrained Java-based systems as it is compositional, to avoid re-verification of already verified bytecode upon loading of new bytecode, and it is regressive, to cleanly support bytecode unloading.

**Key-words:** control flow, static analysis, Java, embedded systems, security

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley, 59650 Villeneuve d'Ascq
Téléphone : +33 3 59 57 78 00 — Télécopie : +33 3 59 57 78 50

# Vérification de politiques de flot de contrôle sur du bytecode Java

**Résumé :** Ce rapport présente l'application de politiques de flot de contrôle sur du bytecode Java pour les petits systèmes ouverts. La plupart du temps, l'application de ce type de politiques de sécurité est réalisée par l'observation du système ou l'insertion de code pour assuré en assurer le respect, ce qui n'est pas approprié pour les petits systèmes fortement contraints tels que les téléphones mobiles ou les cartes à puce. Nous présentons une méthode basée sur le *proof-carrying code* pour faire appliquer ce type de politiques avec une vérification embarquée réalisée au chargement. Notre approche est bien adaptée aux petits systèmes ouverts évolutifs car elle est compositionnelle, pour éviter la revérification du code déjà chargé, et régressive, afin de traiter proprement le déchargement de code déjà installé et vérifié.

**Mots-clés :** flot de controle, analyse statique, Java, systèmes embarqués, sécurité

## 1 Introduction

Ubiquitous devices such as mobile phones and smart-cards are multi-application capable and support post–issuance installation of applications. Applications are also evolving to take advantage of this new trend: they have shifted from standalone designs to a collaborative model where they provide and/or use services of other applications. In this context, operating systems of ubiquitous devices provide few mechanisms to protect themselves, and end-users, against misuses.

In Android for instance, an application can request some permissions at installation such as the ability to read the address book stored in the phone (`READ_CONTACTS`), or to open network sockets (`INTERNET`). The end-user can accept or refuse to grant these permissions to an application, but there is no way to control how the application is going to use them: is it going to send my contacts to a third party? If I refuse to grant a permission $P$ to an application, can it collude with some other application that has been granted $P$?

*Code-signing* was the first technique introduced to protect ubiquitous devices against code originated from untrusted entities, but it does not guarantee any security property on the behavior of the loaded code. Code-signing is now also used to ensure integrity of the code bounded together with some metadata describing its security-related behavior. This is for instance the case in *model-carrying code (MCC)* [27] and *security-by-contract (S×C)* [5,18] models. The embedded metadata are checked at loading time by the device to determine they are compliant with its security policy. However, adequacy between the code and the metadata describing its behavior is enforced at run-time by execution monitoring, which is not appropriate for constrained devices such as smart-cards or mobile phones as it produces variable run-time overheads according to the complexity of the security policy to be enforced. A common approach for the enforcement of security policies using execution monitoring consists in relying on security automata [4,10,26,11,29,7,13]. While security automata provide a very expressive mean to describe security policies, execution monitoring of the security automaton states implies to react before the policy is violated when some misuse is about to occur: execution can be halted [1,26] or control flow can be dynamically altered [16,17,6,28,11] in order to make it compliant with the security policy, but both actions strongly impact applications functionality.

McDaniel *et al* [23] described a context-sensitive security framework for Android to monitor and to restrict applications behavior. It is for example possible to restrict applications to be granted both `READ_CONTACTS` and `INTERNET` permissions, but also for an application to refuse interactions with some application according to its permissions. However, this framework cannot avoid collusion between applications, for instance between one with the `READ_CONTACT` permission, and the another one with the `INTERNET` permission. This approach lacks refinement as it can block the installation of "honest" applications (such as a reliable text messenger application in the previous example) without detecting possibly dangerous misuses. Information flow techniques such as [21] can solve this problem, but they are practically not used because they require an *ad hoc* run-time environment and developers with a strong understanding of the underlying model to smartly annotate their code. To cope with these problems, McDaniel *et al* recently proposed another framework [9] for Android with run-time enforcement of secure information flow. This approach is obviously not portable to devices more constrained than modern phones, *i.e.*, J2ME or Java Card devices.

Static analysis offers a good alternative to execution monitoring as it does not produce computation overheads at run-time and does not alter code's behavior dynamically. However,

static analysis requires high computational and memory resources not available in constrained devices. The *Proof-carrying code (PCC)* model [22] permits to simplify on-device analysis. A *proof* of the code's behavior with respect to a given property is pre-computed off-device so that the code's receiver only needs to verify that the proof is correct for the given code. Verifying this proof is actually far more easier and consumes less resources than doing the complete analysis on-device while it offers the same security guarantees, moreover without the need of a trusted entity for signing the code. Proof-carrying code approaches have already been successfully applied in constrained devices, in particular by Rose [25] on Java bytecode. Existing approaches mainly focus on type safety [20,15,25], space/time guarantees [3], safety properties [2,30,32], access control and control flow policies [14,8,24]. All aforementioned works are not purely static analysis. For instance, Jensen *et al* [14] proposed to enforce a *global* control flow policy through *local* run-time checks present in the code. Their approach consists in verifying that these local checks are sufficient to enforce a global control flow policy. This approach is closely related to [8] where similar in-line checks are generated just-in-time.

In this paper, we propose a technique in-between security automata [26], Jensen *et al* work on the verification of control flow policies [14] and compositional verification of control flow policies described in [12,19]. Our main contribution is to propose a purely static and compositional proof-carrying code approach to enforce global control flow policies specified by an automaton devoted to constrained Java-based devices such as mobile phones or smart cards. This paper is structured as follows. Section 2 introduces the formal notations used in the rest of the paper. Section 3 describes the theoretical foundations of our approach: the definition of a *global control flow policy* of a system, and the definition of a method's *footprint*, that is its relevant contribution to a global policy. Section 4 details the static analysis of Java bytecode needed to compute method footprints off-device, and how method footprints are efficiently encoded. Section 5 details how proof obligations that are method footprints can be efficiently and incrementally verified on-device upon (un)loading of bytecode. Section 6 draws conclusions and presents future work.

## 2 Notations

### 2.1 Object-oriented notations

In this paper, we consider multi-application Java systems. As in any Java system, an application is implemented by a set of classes. However, the notion of application does not correspond to something concrete on most Java sytems. We voluntary use a generic meaning for applications in order to match any Java environment: for traditional Java, an application can simply be a fully qualified package name, while it can correspond to a CAP file on Java Card systems.

We will write $\mathcal{M}$ the set of all method names that are fully qualified names, namely elements of the form $A.C.m$ to denote the method $m$ that is available in objects of class $C$ of application $A$. Note that the method $A.C.m$ might be defined and implemented in a super-class of $A.C$ – which might itself be in some other application – and simply inherited.

We use the following notations for inheritance. The set of classes is equipped with an inheritance relation $\leq$; $C_1 \leq C_2$ means that the class $C_1$ inherits from $C_2$ or is $C_2$ ($<$ for a strict sub-class). The set $\mathcal{M}$ is also equipped with an inheritance relation, also written $\leq$. When a class $C_1$ of application $A_1$ inherits from some class $C_2$ in application $A_2$ and redefines a method $m$, we write $A_1.C_1.m < A_2.C_2.m$. We use $\leq$ whenever the method is either inherited

or redefined. The *method definition* $def(A.C.m)$ of $A.C.m$ is $A.C.m$ if the method $m$ is defined or redefined in $A.C$, otherwise $def(A.C.m) = A'.C'.m$ such that $A.C < A'.C'$ and for each $A''.C''$ such that $A.C < A''.C'' < A'.C'$, $m$ is neither defined nor redefined in $A''.C''$.

### 2.2 Graphs of the programs

We consider finite directed graphs with edges labeled by elements of a set $L_E$ and unlabeled vertices (as we consider only cases where the labeling function of vertices would have been a bijection). A graph $G$ is given by a pair $(V, E)$ where $V$ is its set of vertices, $E \subseteq V \times V \times L_E$, and its set of labeled edges.

In the graph $G = (V, E)$, the edge from the vertex $u$ to $v$, labeled by $l$ is denoted by $(u, v, l)$. Graphs may also have unlabeled edges, which are edges for which labeling is irrelevant and can be ignored, edges of such graphs are written as pairs of vertices $(u, v)$.

For a graph $G = (V, E)$, $V' \subseteq V$ and $V'' \subseteq V$:

$$paths(G, V', V'') = \{v_0 v_1 \dots v_n \in V^* \mid v_0 \in V', v_n \in V'', \forall 0 < i \leq n, (v_{i-1}, v_i) \in E\}.$$

A strongly connected component of a graph $G = (V, E)$ is a subgraph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$, where $V'$ is a maximal subset of $V$ such that for each pair $v_1$, $v_2$ of vertices of $V'$, there exists a word $v_1 v_{i_0} \dots v_{i_n} v_2$ in $paths(G, V', V')$ and each $v_{i_k} \in V'$ (and symmetrically from $v_2$ to $v_1$).

For a method $m$, $P_m$ is its instruction list. We assume this list to be indexed from 0 to $|P_m| - 1$, where $|P_m|$ is its size. Hence, we denote by $P_m[i]$ the $i + 1$-th bytecode instruction of the method $m$.

Let us first define the call graph of a set of methods.

**Definition 1 (Call graph for a set of methods).** *Let $M$ be a set of methods. A call graph for $M$ is a finite graph $CG = (M, E)$, $E \subseteq M \times M \times \mathbb{N}$, such that: for each $m_1 \in M$, for each instruction $P_{m_1}[i] =$* `invoke` [1] *$m_2$, for all $m'_2 \leq m_2$ that might be called at run-time by this instruction $(m_1, m'_2, i) \in E$.* [2]

**Definition 2 (Intraprocedural control flow graph).** *The* intraprocedural control flow graph *of a method $m$ is an unlabeled graph $CF_m = (P_m, E)$ such that $(i, i')$ belongs to $E$ if either* (1) *$i' = i + 1$ and $P_m[i]$ is different from a* `return` *bytecode and from* `goto` *$a$, or* (2) *$i' = a$ and $P_m[i]$ is equal to* `goto` *$a$ or to a comparison (or similar cases) bytecode* `ifcmp` *$a$. Hence, there is no edge going out of the vertex $i$ if $P_{C.m}[i]$ is a* `return` *bytecode. An exit-point in a control flow graph is a vertex without successor.*

To deal with control flow properties, we have to be able to compute for a given method, the set of methods it invokes directly of indirectly. The set is statically computed and thus collects methods that are invoked on static types.

**Definition 3 (Invoked methods).** *Let $m$ be a method. Then the set of methods that $m$ invokes is the smallest set of methods such that $\mathcal{I}(m) = \mathcal{I}^{direct}(m) \cup \mathcal{I}^{indirect}(m)$ with*

- *$\mathcal{I}^{direct}(m) = \{m' \mid \exists 0 \leq i < |P_m|, P_m[i] =$* `invoke` *$m'', m' \leq m''\}$ is the set of methods invoked in the bytecode of $m$,*

[2] `invoke` stands for any invocation bytecode like `invokevirtual`, `invokestatic`, *etc.*

– $\mathcal{I}^{indirect}(m) = \bigcup_{m' \in \mathcal{I}^{direct}(m)} \mathcal{I}(m')$, *i.e., the set of methods (directly and indirectly) invoked by methods directly invoked by* $m$.

It is fairly easy to see that $\mathcal{I}(m)$ is exactly the set of all the methods that are reachable (by 1 or more transitions) from $m$ in the call graph of the system (Definition 1).

### 2.3 Formal languages notations

In this section we give additional notations that are used in this paper for finite automata and languages.

**Definition 4 (Finite automaton).** *A* finite automaton *is a tuple* $A = (\Sigma, S, s_0, \zeta, S_F)$ *where* $\Sigma$ *is the input alphabet,* $S$ *is the set of states,* $s_0 \in S$ *is the initial state,* $\zeta : S \times \Sigma \longrightarrow S$ *is the transition function, and* $S_F \subseteq S$ *is the set of final states.*

**Definition 5 (Trimmed automaton).** *An automaton* $A = (\Sigma, S, s_0, \zeta, S_F)$ *is* trimmed *if for all state* $s \in S$*, there exist two words* $u$ *and* $v$ *such that* $\zeta(s_0, u) = s$ *and* $\zeta(s, v) \in S_F$*.*

**Definition 6 (Language).** *Let* $A = (\Sigma, S, s_0, \zeta, S_F)$ *be a finite automaton. The* language *of* $A$*, denoted by* $\mathcal{L}(A)$ *is defined by:*

$$\mathcal{L}(A) = \{a_0 \dots a_n \in \Sigma^* \mid \exists (s_{i_0}, a_0, s_{i_1})(s_{i_1}, a_1, s_{i_2}) \dots (s_{i_n}, a_n, s_{i_{n+1}}) \\ \forall 0 \leq k \leq n, (s_{i_k}, a_k, s_{i_{k+1}}) \in \zeta, s_{i_0} = s_0, s_{i_{n+1}} \in S_F\}.$$

Note that $\varepsilon$ denotes the empty word. We now define the factors of a language.

**Definition 7 (Factors).** *Let* $L$ *be a language of* $\Sigma^*$*. Then, the* left factors*, (regular)* factors*, and* right factors *of* $L$ *are respectively defined by:*

$$lf(L) = \{u \in \Sigma^* \mid \exists w \in \Sigma^*, uw \in L\},$$
$$fact(L) = \{u \in \Sigma^* \mid \exists v, w \in \Sigma^*, vuw \in L\},$$
$$rf(L) = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, vu \in L\}.$$

**Definition 8 (Projection).** *Let* $\Sigma$ *and* $\Xi$ *be two alphabets. Let* $L \subseteq \Sigma^*$ *be a language. The* projection onto $\Xi$ *is the alphabetical morphism* $\Pi_\Xi$ *from* $\Sigma$ *to* $\Xi$ *such that for each* $x$ *of* $\Sigma$*,* $\Pi_\Xi(x) = x$ *if* $x \in \Xi$ *and* $\Pi_\Xi(x) = \varepsilon$ *otherwise.*

## 3 Global control flow policy of a system

In this work, we study the control flow policies in terms of access to methods. We introduce the notion of *global policy* of a system that defines the control applied on applications by the system. The global policy of a system defines the sequences of method calls that are forbidden. This type of policy can be used for example to restrict applications access to the system API.

### 3.1 Definition of the global policy of a system

At the verification level, the forbidden sequences of calls are described by a finite automaton. This automaton is really simple compared to security automata [26] as it has no vertex labelling to describe a particular state of the system, and an edge label can only describe a method call. Actually, we show along the rest of the paper that these simplifications are useful to achieve a complete static verification technique especially in open and constrained environments.

**Definition 9 (Global policy).** *Let the* global policy *of a system $\mathcal{G}$ be a finite trimmed automaton $\mathcal{G} = (\Sigma, S, s_0, \zeta, \{s_F\})$ with $\Sigma \subseteq \mathcal{M}$ and such that:*

- *there is no $(s, a) \in S \times \Sigma$ with $\zeta(s, a) = s_0$,*
- *there is no $(s, a) \in S \times \Sigma$ with $\zeta(s_F, a) = s$.*

**Definition 10 (Conformity to a policy).** *We say that a system conforms to the policy $\mathcal{G}$ if for each execution trace $t \in \mathcal{M}^*$, $conform(t, \mathcal{G})$ holds with:*

$$conform(t, \mathcal{G}) \Leftrightarrow (\forall v \text{ such that } \Pi_\Sigma(t) = uvu', v \notin \mathcal{L}(\mathcal{G})).$$

Note that we prohibit any transition incoming into the initial state and outgoing of the final state in our global policies. Since a trace is not conforming as soon as it contains a word of the language of the global policy, transitions starting in a final state or leading to the initial state are useless to detect non-conforming traces: on the one hand, the trace was already recognized as invalid the first time $s_F$ was reached and on the other hand we can simply chop off the prefix corresponding to the loop from $s_0$ to $s_0$ while keeping an invalid trace.

Also note that, only one final state is needed since this state has no outgoing transition: if there were various final states they would be equivalent and could be merged. Consequently, we write in the rest of this document $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$.

### 3.2 Global policy footprint of a method

In this work, we consider open systems that support dynamic application loading. Thus, we aim at a compositional model in which methods can be verified one by one and systems can be extended with new methods without re-verification of already loaded code. In this system, a method $m$ can be valid for a global policy but still contain a part of an invalid trace. If this method is invoked by another one that produces the beginning of a forbidden trace, invokes $m$, and produces the end of a forbidden trace, then $m$ might participate in the construction of an invalid trace even if all the traces of $m$ are allowed. To track this kind of behaviors, we define in this section the contribution of a method to the current execution with respect to the global policy of a system; we call that contribution the *footprint* of the method.

**Definition 11 (Interprocedural control flow graph).** *The interprocedural control flow graph of a set of methods $M$ is the graph $ICFG_M = (V, E)$, built from the intraprocedural control flow graphs $CF_m$ of the methods $m \in M$, with $def(M) = M \cup \{def(m) \mid m \in M\}$, such that:*

- $V = \{m.i \mid i \in V_m, m \in def(M), CF_m = (V_m, E_m)\}$,
- $E = \{(m.i, m.i') \mid m \in def(M) \wedge CF_m = (V_m, E_m) \wedge (i, i') \in E_m \wedge P_m[i] \neq \texttt{invoke}\ m' \text{ with } m' \in def(M)\}$
  $\cup \{(m.i, m''.0) \mid P_m[i] = \texttt{invoke}\ m' \wedge m' \in def(M) \wedge m'' \leq def(m') \wedge m'' = def(m'')\}$
  $\cup \{(m.i, m'.i') \mid P_m[i] = \texttt{return} \wedge P_{m'}[i' - 1] = \texttt{invoke}\ m'' \wedge m \leq def(m'')\}$

Let $m$ be a method. Let us remind that $\mathcal{I}_m$ is the set of methods called by a method $m$, directly or indirectly. Then the interprocedural control flow graph of $m$ is $ICFG_m = ICFG_{\{m\} \cup \mathcal{I}_m}$.

In general, it is not possible to compute statically the exact set of traces of a system or a subsystem, so we compute an over-approximation of that set. In particular, we put in the interprocedural control flow graph the edges between an `invoke` instruction and all the methods that could be actually invoked, due to method overloading.

As we only consider method calls in the policy, we define a morphism for $V^*$ to the set $\mathcal{L}(\mathcal{M})$ of languages defined on $\mathcal{M}$ that allows us to restrict the traces to method calls, taking care of method definitions to avoid an attacker to bypass the control by overloading a class:

$$\begin{aligned} \mathit{calls} : V &\longrightarrow \mathcal{L}(\mathcal{M}) \\ m.i &\longmapsto \begin{cases} \{m'' \mid m' \leq m'' \wedge \mathit{def}(m'') = \mathit{def}(m')\} & \text{if } P_m[i] = \texttt{invoke } m' \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

We now define the set of traces of a method $m$, that is an over-approximation of its set of execution traces as:

$$\mathit{traces}(m) = m.\mathit{calls}(\mathit{paths}(\mathit{ICFG}_m, \{m.0\}, \{m.i \mid P_m[i] = \texttt{return}\})).$$

We can now define the $\mathcal{G}$-footprint of a method $m$ for a policy $\mathcal{G}$ that describes the factors (left, regular and right) of the traces of the policy language that may result from the execution of this method.

**Definition 12 ($\mathcal{G}$-footprint of a method).** *The $\mathcal{G}$-footprint of a method $m$ for a policy $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$ is $\mathit{foot}_{\mathcal{G}}(\mathit{traces}(m))$ with*

$$\mathit{foot}_{\mathcal{G}}(L) = \begin{cases} \maltese & \text{if } \mathit{fact}(L) \cap \mathcal{L}(\mathcal{G}) \neq \emptyset \\ (\ \mathit{lf}(\Pi_\Sigma(L)) \cap \mathit{rf}(\mathcal{L}(\mathcal{G})), & \\ \quad \Pi_\Sigma(L) \cap \mathit{fact}(\mathcal{L}(\mathcal{G})), & \text{otherwise} \\ \quad \mathit{rf}(\Pi_\Sigma(L)) \cap \mathit{lf}(\mathcal{L}(\mathcal{G}))\ ) & \end{cases}$$

*We denote by $\mathcal{F}_{\mathcal{G}}$ the set of footprints for a policy $\mathcal{G}$.*

This definition allows us to keep information about the contribution of a method $m$ to creation of forbidden sequences as factors. That contribution is written ✠ whenever the method contains an execution trace that is prohibited. Otherwise, it is described as a tuple: the first element of the tuple contains the possible ends of forbidden traces (left factors of complete execution traces of the method); the second element contains the full execution traces of the method that are middle elements of forbidden traces, and the last element of the tuple contains the possible beginnings of forbidden traces. All these factors will be later aggregated with the beginnings and ends of traces of a method that invokes $m$ in order to produce the footprint of this calling method.

### 3.3 Properties of the operations on footprints

In this section, we present the main operations on footprints and their properties, that will be useful to obtain compositionality.

**Definition 13 (Union of $\mathcal{G}$-footprints).** *Let $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$ be a global policy and $f_1$ and $f_2$ be two $\mathcal{G}$-footprints. Then the* union of $f_1$ and $f_2$ *is defined depending on their forms:*

$$\begin{aligned}
\maltese \cup \maltese &= \maltese \\
\maltese \cup (LF_2, F_2, RF_2) &= \maltese \\
(LF_1, F_1, RF_1) \cup \maltese &= \maltese \\
(LF_1, F_1, RF_1) \cup (LF_2, F_2, RF_2) &= (LF_1 \cup LF_2, F_1 \cup F_2, RF_1 \cup RF_2)
\end{aligned}$$

**Lemma 1 ($foot_{\mathcal{G}}$ distributes over union).** *Let $L_1$ and $L_2$ be two languages, then we have $foot_{\mathcal{G}}(L_1 \cup L_2) = foot_{\mathcal{G}}(L_1) \cup foot_{\mathcal{G}}(L_2)$.*

*Proof.* If one of the languages contains a word with a prohibited factor, its footprint will be $\maltese$, as will be the union of the footprints and the footprint of the union.
Otherwise, the result follows from the fact that the set of factors (resp. left or right) of a language is the set containing all the factors (resp. left or right) of its words. □

**Definition 14 (Concatenation of $\mathcal{G}$-footprints).** *Let $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$ be a global policy. We define the* concatenation of two $\mathcal{G}$-footprints *depending on their forms.*

$$\begin{aligned}
\maltese.\maltese &= \maltese \\
(LF_1, F_1, RF_1).\maltese &= \maltese \\
\maltese.(LF_2, F_2, RF_2) &= \maltese \\
(LF_1, F_1, RF_1).(LF_2, F_2, RF_2) &= \begin{cases}
\maltese & \text{if } RF_1.LF_2 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset \\
(\ LF_1 \cup (F_1.LF_2 \cap rf(\mathcal{L}(\mathcal{G}))), & \\
\quad F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G})), & \text{otherwise} \\
\quad RF_2 \cup (RF_1.F_2 \cap lf(\mathcal{L}(\mathcal{G})))\ ) &
\end{cases}
\end{aligned}$$

Clearly, the concatenation is not commutative, but it has the following property:

**Lemma 2.** *The concatenation of footprints is associative.*

*Proof.* Let us consider three footprints. If one of them is $\maltese$, the result is immediate.
Otherwise, we want to show that

$$((L_1, F_1, R_1).(L_2, F_2, R_2)).(L_3, F_3, R_3) = (L_1, F_1, R_1).((L_2, F_2, R_2).(L_3, F_3, R_3)).$$

Let us give names to the two sides:

$$\begin{aligned}
Res_1 &= ((L_1, F_1, R_1).(L_2, F_2, R_2)).(L_3, F_3, R_3) \\
Res_2 &= (L_1, F_1, R_1).((L_2, F_2, R_2).(L_3, F_3, R_3))
\end{aligned}$$

We have $(L_1, F_1, R_1).(L_2, F_2, R_2) = \maltese$ only when $R_1.L_2 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$. On the other side, if $(L_2, F_2, R_2).(L_3, F_3, R_3) = \maltese$, the results will obviously coincide. Otherwise, the left factor of $(L_2, F_2, R_2).(L_3, F_3, R_3)$ will contain $L_2$ so $(L_1, F_1, R_1).((L_2, F_2, R_2).(L_3, F_3, R_3))$ will be forced to be $\maltese$ too. The converse is similar.
Let us develop further if $(L_1, F_1, R_1).(L_2, F_2, R_2) \neq \maltese$ and $(L_2, F_2, R_2).(L_3, F_3, R_3) \neq \maltese$.

$$\begin{aligned}
Res_1 &= (L_1 \cup (F_1.L_2 \cap rf(\mathcal{L}(\mathcal{G}))), F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G})), R_2 \cup (R_1.F_2 \cap lf(\mathcal{L}(\mathcal{G}))).(L_3, F_3, R_3) \\
Res_2 &= (L_1, F_1, R_1).(L_2 \cup (F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G}))), F_2.F_3 \cap fact(\mathcal{L}(\mathcal{G})), R_3 \cup (R_2.F_3 \cap lf(\mathcal{L}(\mathcal{G}))))
\end{aligned}$$

If $Res_1 = \maltese$ then $(R_2 \cup R_1.F_2).L_3 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$. Since $(L_2, F_2, R_2).(L_3, F_3, R_3) \neq \maltese$, that would mean $R_1.F_2.L_3 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$. This would mean that $R_1.(F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G}))) \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$ so $Res_2 = \maltese$. The converse is similar.
Assuming that $Res_1 \neq \maltese$ and $Res_2 \neq \maltese$, we finish developing the two terms.

$$\begin{aligned}
Res_1 = ((&L_1 \cup (F_1.L_2 \cap rf(\mathcal{L}(\mathcal{G})))) \cup ((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3 \cap rf(\mathcal{L}(\mathcal{G}))),\\
&(F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).F_3 \cap fact(\mathcal{L}(\mathcal{G})),\\
&A)\\
Res_2 = (&L_1 \cup (F_1.(L_2 \cup (F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G})))) \cap rf(\mathcal{L}(\mathcal{G}))),\\
&F_1.(F_2.F_3 \cap fact(\mathcal{L}(\mathcal{G}))) \cap fact(\mathcal{L}(\mathcal{G})),\\
&B)
\end{aligned}$$

We first consider factors (second component) of $Res_1$ and $Res_2$: $(F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).F_3 \cap fact(\mathcal{L}(\mathcal{G}))$ and $F_1.(F_2.F_3 \cap fact(\mathcal{L}(\mathcal{G}))) \cap fact(\mathcal{L}(\mathcal{G}))$. In both cases, if $u$ is a word of the language, $u$ is a word of $fact(\mathcal{L}(\mathcal{G}))$ that is composed of three factors: $u = u_1.u_2.u_3$ with $u_i \in F_i$, as $fact(\mathcal{L}(\mathcal{G}))$ is closed by $fact$, we get the equality.

We now consider the left factors (first component) of $Res_1$ and $Res_2$:

$$\begin{aligned}
&(L_1 \cup (F_1.L_2 \cap rf(\mathcal{L}(\mathcal{G})))) \cup ((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3 \cap rf(\mathcal{L}(\mathcal{G})))\\
&= L_1 \cup (F_1.L_2 \cap rf(\mathcal{L}(\mathcal{G}))) \cup ((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3 \cap rf(\mathcal{L}(\mathcal{G})))
\end{aligned}$$

and

$$L_1 \cup (F_1.(L_2 \cup (F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G})))) \cap rf(\mathcal{L}(\mathcal{G}))).$$

So we have to show that

$$\begin{aligned}
&(F_1.L_2 \cap rf(\mathcal{L}(\mathcal{G}))) \cup ((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3 \cap rf(\mathcal{L}(\mathcal{G})))\\
&= ((F_1.L_2) \cup ((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3)) \cap rf(\mathcal{L}(\mathcal{G}))
\end{aligned}$$

and

$$\begin{aligned}
&F_1.(L_2 \cup (F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G})))) \cap rf(\mathcal{L}(\mathcal{G}))\\
&= (F_1.L_2 \cup F_1.(F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G})))) \cap rf(\mathcal{L}(\mathcal{G}))
\end{aligned}$$

are equal. That can be reduced to show that $((F_1.F_2 \cap fact(\mathcal{L}(\mathcal{G}))).L_3) \cap rf(\mathcal{L}(\mathcal{G}))$ and $(F_1.(F_2.L_3 \cap rf(\mathcal{L}(\mathcal{G})))) \cap rf(\mathcal{L}(\mathcal{G}))$ are equal. In both cases if $u$ is a word of the language, it is composed of three factors: $u = u_1.u_2.u_3$ with $u_1 \in F_1$, $u_2 \in F_2$, and $u_3 \in L_3$. As $u$ belongs to $rf(\mathcal{L}(\mathcal{G}))$, obviously $u_2.u_3 \in rf(\mathcal{L}(\mathcal{G}))$, and $u_1u_2 \in fact(\mathcal{L}(\mathcal{G}))$.

□

We now show that $foot_{\mathcal{G}}$ is an endomorphism, *i.e.*, it somehow "distributes" over concatenation.

**Lemma 3 (Footprint is a morphism).** *Let $L_1$ and $L_2$ be two languages, then we have $foot_{\mathcal{G}}(L_1.L_2) = foot_{\mathcal{G}}(L_1).foot_{\mathcal{G}}(L_2)$.*

*Proof.* If $foot_{\mathcal{G}}(L_1) = \maltese$ or $foot_{\mathcal{G}}(L_2) = \maltese$, then $foot_{\mathcal{G}}(L_1.L_2) = \maltese$ because $fact(L_i) \subseteq fact(L_1.L_2)$.
Otherwise, let $foot_{\mathcal{G}}(L_1) = (LF_1, F_1, RF_1)$ and $foot_{\mathcal{G}}(L_2) = (LF_2, F_2, RF_2)$. If we have $foot_{\mathcal{G}}(L_1.L_2) = \maltese$, we know that there is a factor of $L_1.L_2$ in $\mathcal{L}(\mathcal{G})$. That factor must be of the form $u_1.u_2$, with $u_1 \in rf(\Pi_\Sigma(L_1))$, $u_2 \in lf(\Pi_\Sigma(L_2))$. Obviously, $u_1 \in lf(\mathcal{L}(\mathcal{G}))$ and $u_2 \in rf(\mathcal{L}(\mathcal{G}))$. So $u_1 \in RF_1$ and $u_2 \in LF_2$ which implies that $RF_1.LF_2 \cap \mathcal{L}(\mathcal{G}) = \emptyset$ so $foot_{\mathcal{G}}(L_1).foot_{\mathcal{G}}(L_2) = \maltese$.
If $foot_{\mathcal{G}}(L_1.L_2) \neq \maltese$, we have:

$$\begin{aligned} foot_{\mathcal{G}}(L_1.L_2) = (&lf(\Pi_\Sigma(L_1.L_2)) \cap rf(\mathcal{L}(\mathcal{G})),\\ &\Pi_\Sigma(L_1.L_2) \cap fact(\mathcal{L}(\mathcal{G})),\\ &rf(\Pi_\Sigma(L_1.L_2)) \cap lf(\mathcal{L}(\mathcal{G}))) \end{aligned}$$

Let us first consider the factors :

$$\begin{aligned} &\Pi_\Sigma(L_1.L_2) \cap fact(\mathcal{L}(\mathcal{G}))\\ =\ &(\Pi_\Sigma(L_1).\Pi_\Sigma(L_2)) \cap fact(\mathcal{L}(\mathcal{G}))\\ =\ &((\Pi_\Sigma(L_1) \cap fact(\mathcal{L}(\mathcal{G}))).(\Pi_\Sigma(L_2) \cap fact(\mathcal{L}(\mathcal{G})))) \cap fact(\mathcal{L}(\mathcal{G})) \end{aligned}$$

For left factors we have:

$$\begin{aligned} &lf(\Pi_\Sigma(L_1.L_2)) \cap rf(\mathcal{L}(\mathcal{G}))\\ =\ &lf(\Pi_\Sigma(L_1).\Pi_\Sigma(L_2)) \cap rf(\mathcal{L}(\mathcal{G}))\\ =\ &(lf(\Pi_\Sigma(L_1)) \cup \Pi_\Sigma(L_1).lf(\Pi_\Sigma(L_2))) \cap rf(\mathcal{L}(\mathcal{G}))\\ =\ &(lf(\Pi_\Sigma(L_1)) \cap rf(\mathcal{L}(\mathcal{G}))) \cup (\Pi_\Sigma(L_1).lf(\Pi_\Sigma(L_2)) \cap rf(\mathcal{L}(\mathcal{G})))\\ =\ &LF_1 \cup (\Pi_\Sigma(L_1).lf(\Pi_\Sigma(L_2)) \cap rf(\mathcal{L}(\mathcal{G}))) \end{aligned}$$

As all the words of $rf(\mathcal{L}(\mathcal{G})))$ are concatenations of factors of $\mathcal{L}(\mathcal{G}))$

$$\begin{aligned} &= LF_1 \cup ((\Pi_\Sigma(L_1) \cap fact(\mathcal{L}(\mathcal{G}))).lf(\Pi_\Sigma(L_2)) \cap rf(\mathcal{L}(\mathcal{G})))\\ &= LF_1 \cup (F_1.lf(\Pi_\Sigma(L_2)) \cap rf(\mathcal{L}(\mathcal{G}))) \end{aligned}$$

Obviously, we have:

$$\begin{aligned} &= LF_1 \cup (F_1.(lf(\Pi_\Sigma(L_2)) \cap rf(\mathcal{L}(\mathcal{G}))) \cap rf(\mathcal{L}(\mathcal{G})))\\ &= LF_1 \cup (F_1.LF_2 \cap rf(\mathcal{L}(\mathcal{G}))) \end{aligned}$$

and we can obtain the right factors in the same way. □

From Lemma 1 and Lemma 3, we get the following proposition:

**Proposition 1.** *For a language $L \subseteq \mathcal{M}^*$ and a global policy of the system $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$, we have*

$$foot_{\mathcal{G}}(L) = \bigcup_{\substack{m_0 m_1 \dots m_n \in L \mid \\ \forall 0 \leq i \leq n, m_i \in \mathcal{M}}} foot_{\mathcal{G}}(\{m_0\}).foot_{\mathcal{G}}(\{m_1\}) \dots foot_{\mathcal{G}}(\{m_n\}).$$

### 3.4 Compositionality of the footprint computation

The $\mathcal{G}$-footprints of methods can be computed in a compositional way except in the presence of mutual recursive methods that have to be analyzed together. For a method $m$, we compute the strongly connected components of the graph $CG_m$. Starting from $m$, we have a partial order of components as the transitive closure of the relation saying that a component $c_1$ is lower than a component $c_2$ if there exists an edge from $c_1$ to $c_2$. Then, the methods of each strongly connected component have to be analyzed together, when all the greater components have been analyzed (*i.e.*, when their footprints are available). The proof of the next proposition is straightforward from Proposition 1.

**Proposition 2 (Compositionality).** *Let us consider a method $m$ and a global policy of the system $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$, and $M$ the set of methods of the strongly connected component of $CG_m$ that contains $m$. Then,*

$$foot_{\mathcal{G}}(traces(m)) = compose_{\mathcal{G},M}(calls(paths(ICFG_M, m.0, \{m.i \mid P_m[i] = \texttt{return}\})))$$

*with the morphism*

$$compose_{\mathcal{G},M} : \mathcal{M} \longrightarrow \mathcal{F}_{\mathcal{G}}$$
$$m' \longmapsto \begin{cases} foot_{\mathcal{G}}(\{m'\}) & \text{if } m' \in M \\ foot_{\mathcal{G}}(traces(m')) & \text{otherwise} \end{cases}$$

## 4 Implementation of the footprint computation

Since we target small systems, we need to provide a compact representation of footprints that uses as little memory as possible and that is easy to manipulate. For this purpose, we use sets of pairs of automaton states to represent footprints. Using this representation, we only need one set to describe one footprint since left factors obviously end at $s_F$ and right factors obviously start at $s_0$.

**Definition 15 ($\mathcal{G}$-footprint implementation).** *Let $\mathcal{G}$ be a global policy of the system. The* $\mathcal{G}$-footprint implementation *is given by the function:*

$$\begin{aligned} \Phi : \quad \mathcal{F}_{\mathcal{G}} &\longrightarrow \wp(S \times S) \\ \maltese &\longmapsto \{(s_i, s_j) \mid \forall s_i, s_j \in S\} \\ (LF, F, RF) &\longmapsto \{(s_i, s_j) \mid \exists u \in F, \zeta(s_i, u) = s_j\} \\ &\quad \cup \{(s_0, s_i) \mid \exists u \in RF, \zeta(s_0, u) = s_i\} \\ &\quad \cup \{(s_i, s_F) \mid \exists u \in LF, \zeta(s_i, u) = s_F\} \end{aligned}$$

*We write $F_m$ for the implementation of the $\mathcal{G}$-footprint of a method $m$, namely $F_m = \Phi(foot_{\mathcal{G}}(traces(m)))$.*

We now define the composition of footprint implementations.

**Definition 16 (Composition).** *The composition over $\wp(S \times S)$ is defined as, for $S_1$ and $S_2$:*

$$\begin{aligned} S_1 \oplus S_2 = &\{(s_i, s_F) \mid (s_i, s_F) \in S_1\} \\ &\cup \{(s_0, s_i) \mid (s_0, s_i) \in S_2\} \\ &\cup \{(s_i, s_j) \mid \exists k, (s_i, s_k) \in S_1, (s_k, s_j) \in S_2 \\ &\qquad \text{or } \exists k, (s_0, s_k) \in S_1, (s_k, s_F) \in S_2 \\ &\qquad \text{or } (s_0, s_F) \in S_1 \cup S_2\} \end{aligned}$$

Note that the definition forces the composition to be the full set $S \times S$ as soon as it contains $(s_0, s_F)$. It is easy to see that $\{(s_i, s_j) \mid \forall s_i, s_j \in S\}$ is absorbing for $\oplus$: this corresponds to the element ✠ for normal footprints. So we will also write ✠ for the full set $S \times S$.

**Lemma 4.** *Let $S_1$, $S_2$, $S_3$ and $S_4$ be elements of $\wp(S \times S)$. The composition is monotonic: if $S_1 \subseteq S_2$ and $S_3 \subseteq S_4$ then $S_1 \oplus S_3 \subseteq S_2 \oplus S_4$.*

**Lemma 5.** *$\Phi$ is a morphism, i.e., $\Phi(f_1.f_2) = \Phi(f_1) \oplus \Phi(f_2)$.*

*Proof.* If $f_i = ✠$, then $f_1.f_2 = ✠$ and $\Phi(f_i) = \Phi(f_1.f_2) = ✠$ which will absorb $\Phi(f_{3-i})$.
Let us then assume that $f_1 \neq ✠$ and $f_2 \neq ✠$. We write $f_1 = (LF_1, F_1, RF_1)$ and $f_2 = (LF_2, F_2, RF_2)$.
Let us first prove that $\Phi(f_1.f_2) \subseteq \Phi(f_1) \oplus \Phi(f_2)$. If $f_1.f_2 = ✠$, this means that $RF_1.LF_2 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$. So we have some $u = u_1.u_2$ such that $u_1 \in RF_1$ and $u_2 \in LF_2$ and $u \in \mathcal{L}(\mathcal{G})$, which means that $\zeta(s_0, u) = \zeta(\zeta(s_0, u_1), u_2) = s_F$. From this, we conclude that $(s_0, \zeta(s_0, u_1)) \in \Phi(f_1)$ and $(\zeta(s_0, u_1), s_F) \in \Phi(f_2)$ and so that $\Phi(f_1) \oplus \Phi(f_2) = ✠$ since it contains $(s_0, s_F)$.
Let us now assume that $f_1.f_2 \neq ✠$ and write $f_1.f_2 = (LF, F, RF)$.
We consider each of the three sub-sets composing $\Phi(f_i)$ separately.
We have $F = F_1.F_2 \cap \mathit{fact}(\mathcal{L}(\mathcal{G}))$. Let us consider some $u \in F$. We know that there exist $u_1$ and $u_2$ such that $u = u_1.u_2$, with $u_1 \in F_1$ and $u_2 \in F_2$. If we consider a pair of states $(s_i, s_j)$ such that $\zeta(s_i, u) = s_j$, we know that $\zeta(s_i, u_1)$ is some state $s_k$ and that $(s_i, s_k)$ must be in $\Phi(f_1)$ and that $(s_k, s_j)$ must be in $\Phi(f_2)$ since $\zeta(s_k, u_2) = s_j$, by definition of $\Phi$. So $(s_i, s_j) \in \Phi(f_1) \oplus \Phi(f_2)$.
We have $LF = LF_1 \cup (F_1.LF_2 \cap \mathit{rf}(\mathcal{L}(\mathcal{G})))$. Let us consider some $u \in LF$ and $(s_i, s_F)$ a pair of states such that $\zeta(s_i, u) = s_F$. If $u \in LF_1$, then $(s_i, s_F)$ must be in $\Phi(f_1)$ by definition of $\Phi$; and so it is preserved by $\oplus$. Otherwise, we must have $u \in F_1.LF_2$ so $u = u_1.u_2$ with $u_1 \in F_1$ and $u_2 \in LF_2$. So $\zeta(s_i, u_1)$ must be some state $s_j$ with $\zeta(s_j, u_2) = s_F$. By definition of $\Phi$, we have $(s_i, s_j) \in \Phi(f_1)$ and $(s_j, s_F) \in \Phi(f_2)$ which entails that $(s_i, s_F)$ is in $\Phi(f_1) \oplus \Phi(f_2)$.
By a similar argument we prove that the image of $RF$ is also included in $\Phi(f_1) \oplus \Phi(f_2)$.
Conversely, let us prove that $\Phi(f_1) \oplus \Phi(f_2) \subseteq \Phi(f_1.f_2)$. Let us consider for this $(s_i, s_j)$ in $\Phi(f_1) \oplus \Phi(f_2)$. We reason over the three definition cases of $\oplus$.

1. $s_j = s_F$ and $(s_i, s_F) \in \Phi(f_1)$; then there exists some $u$ such that $\zeta(s_i, u) = s_F$ by definition of $\Phi$ and so $u$ must be in $RF_1 \cup F_1 \cup LF_1$. In all three sub-cases, since it is a right factor of $\mathcal{L}(\mathcal{G})$ (because it can end in $s_F$) it must also be in $LF_1$. This entails that it is in $LF$ by definition of concatenation so $(s_i, s_F)$ is in $\Phi(f_1.f_2)$.
2. $s_i = s_0$ and $(s_0, s_j) \in \Phi(f_2)$. This case is similar to the previous one.
3. We have again three sub-cases.
   - There exists some $s_k$ such that $(s_i, s_k) \in \Phi(f_1)$ and $(s_k, s_j) \in \Phi(f_2)$. So we can find $u_1 \in RF_1 \cup F_1 \cup LF_1$ such that $\zeta(s_i, u_1) = s_k$ and $u_2 \in RF_2 \cup F_2 \cup LF_2$ such that $\zeta(s_k, u_2) = s_j$.
     - If $u_1 \in LF_1$, then $s_j = s_k = s_F$ and $u_2$ must be $\varepsilon$ since there is no transition out of $s_F$. Then $u = u_1$ is in $LF$ and $(s_i, s_F)$ is in $\Phi(f_1.f_2)$.
     - If $u_2 \in RF_2$, then $s_i = s_k = s_0$ and $u_1$ must be $\varepsilon$ since there is no transition to $s_0$. Then $u = u_2$ is in $RF$ and $(s_0, s_j)$ is in $\Phi(f_1.f_2)$.
     - If $u_1 \in F_1$ and $u_2 \in LF_2$, then $s_j = s_F$ and we just need to show that $u_1.u_2$ is in $\mathit{rf}(\mathcal{L}(\mathcal{G}))$ to prove that $u_1.u_2$ is in $LF$. Since $\mathcal{G}$ is trimmed, there must exist some word $v$ such that $\zeta(s_0, v) = s_i$ so $u_1.u_2$ is indeed a right factor of $\mathcal{L}(\mathcal{G})$, which proves that $(s_i, s_F)$ is in $\Phi(f_1.f_2)$.

  - If $u_1 \in F_1$ and $u_2 \in F_2$, since the automaton is trimmed, $u_1.u_2$ is a factor of $\mathcal{L}(\mathcal{G})$ and is in the factors of $f_1.f_2$ so $(s_i, s_j)$ is in $\Phi(f_1.f_2)$.
  - If $u_1 \in RF_1$ and $u_2 \in LF_2$ then $u_1.u_2 \in \mathcal{L}(\mathcal{G})$ so $RF_1.LF_2 \cap \mathcal{L}(\mathcal{G}) \neq \emptyset$ so $f_1.f_2 = \maltese$ in which case $(s_i, s_j) \in \Phi(f_1.f_2) = \maltese$.
  - If $u_1 \in RF_1$ and $u_2 \in F_2$, then $s_i = s_0$ and there must exist some word $v$ such that $\zeta(\zeta(s_0, u_1.u_2), v) = s_F$ since $\mathcal{G}$ is trimmed. So $u = u_1.u_2$ is a left factor of $\mathcal{L}(\mathcal{G})$, so $u$ must be in the right factors of $f_1.f_2$ which entails that $(s_0, sj) \in \Phi(f_1.f_2)$.

– If $(s_i, s_j)$ is present because there is some $k$ such that $(s_0, s_k) \in \Phi(f_1)$ and $(s_k, s_F) \in \Phi(f_2)$ this means that there is some word $u_1$ such that $\zeta(s_0, u_1) = s_k$ with $u_1$ in the right factors of $f_1$ (because the factors of $f_1$ that are in $lf(\mathcal{L}(\mathcal{G}))$ are also in the right factors; and if $u_1$ is in the left factors, then $s_k = s_F$ so it is also in the right factors) and some $u_2$ such that $\zeta(s_k, u_2) = s_F$ with $u_2$ in the left factors of $f_2$, so $f_1.f_2 = \maltese$ and $(s_i, s_j) \in \Phi(f_1.f_2)$.
– Lastly, if $(s_0, s_F)$ is in $\Phi(f_1)$, then we must have $f_1 = \maltese$ by definition of footprints, since it means that a factor of the underlying language of traces is prohibited. Then $f_1.f_2 = \maltese$ and $\Phi(f_1.f_2) = \maltese$ so $(s_i, s_j) \in \Phi(f_1.f_2)$.

□

$\Phi$ also distributes over $\cup$.

**Lemma 6.** *For any $f_1$ and $f_2$ two footprints, $\Phi(f_1 \cup f_2) = \Phi(f_1) \cup \Phi(f_2)$.*

*Proof.* Let us consider $f_1$ and $f_2$ two footprints. If $f_1 = \maltese$, $\Phi(f_1 \cup f_2) = \Phi(\maltese) = \maltese = \maltese \cup \Phi(f_2)$. The result is identical if $f_2 = \maltese$. If neither of them is $\maltese$, we write $f_1 = (LF_1, F_1, RF_1)$ and $f_2 = (LF_2, F_2, RF_2)$. $f_1 \cup f_2 = (LF_1 \cup LF_2, F_1 \cup F_2, RF_1 \cup RF_2)$ and the definition of $\Phi$ allows us to conclude. □

In the rest of this section, we define a system of equations that allows us to compute the footprint implementation of a method that corresponds to the $\mathcal{G}$-footprint of a method $m$.

For each method $m$ we consider $M$ the set of methods of the strongly connected component of $CG_m$ that contains $m$. We define the system of equations $(S_M)$ where the rules *instr* are given on Figure 1:

$$S_{m''.i} = \bigcup_{(m'.j, m''.i) \in ICFG_M} instr_{P_{m'}[j]}(S_{m'.j})$$

with the initial state

$$S_{m.0} \supseteq \begin{cases} \{(s_i, s_i) \mid s_i \in S\} \cup \{(s_i, s_j) \mid s_i, s_j \in S \text{ if } s_0 = s_F\} & \text{when } m \notin \Sigma \\ \{(s_i, s_j) \mid s_i, s_j \in S, \zeta(s_i, m) = s_j \text{ or } \zeta(s_0, m) = s_F\}\} & \text{when } m \in \Sigma \end{cases}$$

Note that the last rule of Figure 1 takes the union of the footprint implementations of all the actual methods that could be invoked by the instruction.

**Proposition 3.** *The system of equations $(S_M)$ admits a least solution.*

*Proof.* The set $(\wp(S \times S), \subseteq, \cup, \cap)$ is a finite lattice. Lemma 4 implies that the transfer functions $instr_b$ are monotonic with respect to $\subseteq$. Thus we can apply Knaster-Tarski theorem. □

$$\frac{b \neq \texttt{invoke}}{instr_b(S) = S} \qquad \frac{b = \texttt{invoke}\ m' \text{ and } m' \in \Sigma \text{ and } m' \in M}{instr_b(S) = S \oplus \Phi(foot_{\mathcal{G}}(\{m'\}))}$$

$$\frac{b = \texttt{invoke}\ m' \text{ and } m' \notin \Sigma \text{ and } m' \in M}{instr_b(S) = S} \qquad \frac{b = \texttt{invoke}\ m' \text{ and } m' \notin M}{instr_b(S) = S \oplus \bigcup_{m'' \leq m'} F_{m''}}$$

**Fig. 1.** Transfer function $instr_b(S)$.

Finally we prove that this least fix-point construction implements footprints.

**Proposition 4.** *Let $m$ be a method, $M$ the set of methods of the strongly connected component of $CG_m$ that contains $m$. Then*

$$F_m = \bigcup_{i|P_m[i]=\texttt{return}} S_i$$

*with $S_i$ the least solutions of the equations of $(S_M)$.*

Note that this property justifies the last rule of Figure 1: the computation of the solution of $(S_M)$ can be based on the footprint implementations of all the methods that are not in the same strongly connected component of the call graph since that will result in the same value. And this proves that the system of equations indeed provides a way to compute the footprint implementations.

*Proof.* Let us write

$$U = \bigcup_{i|P_m[i]=\texttt{return}} S_i$$

and let us first prove that $F_m \subseteq U$.

$$\begin{aligned} F_m &= \Phi(foot_{\mathcal{G}}(traces(m))) \\ &= \Phi(\bigcup_{t \in traces(m)} foot_{\mathcal{G}}(\{t\})) \\ &= \Phi(\bigcup_{m_1 \ldots m_n \in traces(m)} foot_{\mathcal{G}}(\{m_1\}) \ldots foot_{\mathcal{G}}(\{m_n\})) \\ &= \bigcup_{m_1 \ldots m_n \in traces(m)} \Phi(foot_{\mathcal{G}}(\{m_1\})) \oplus \cdots \oplus \Phi(foot_{\mathcal{G}}(\{m_n\})) \end{aligned}$$

Let us consider any pair in $F_m$ and some trace $t$ of $m$ so that the pair is in $\Phi(foot_{\mathcal{G}}(\{t\}))$. Since traces come from actual execution paths, we consider a path $p$ in $CG_m$ such that it produces $t$. Let us sum up the constraints on $(S_M)$ we obtain by looking at that path $p$. $p$ can be written as $(m_{j_0}.i_{j_0}) \ldots (m_{j_q}.i_{j_q})$ where at each step $j_k$ the instruction $i_{j_k}$ of method $m_{j_k}$ is executed.

Let us reason on the length of $p$ to show that, if we consider a prefix path $p'$ of $p$ of any length, the set $S_{m'.i'}$ must contain $\Phi(foot_{\mathcal{G}}(t'))$ where $t'$ is the part of $t$ corresponding to $p'$.

If $m \in \Sigma$, every trace of $m$ begins with $m$. In that case we also know that $\Phi(foot_{\mathcal{G}}(\{m\})) \subseteq S_{m.0}$ by definition of the initial state of the system $(S_M)$. Otherwise, $S_{m.0}$ initially contains simply $\Phi(foot_{\mathcal{G}}(\{\varepsilon\}))$.

Let us add one instruction to $p'$. For all instruction $P_{m_{j_k}}[i_{j_k}]$ along $p$ that is not an `invoke`, we simply learn that $S_{m_{j_k}.i_{j_k}} \supseteq S_{m_{j_k-1}.i_{j_k-1}}$ by the rules of Figure 1. Correspondingly, $t'$ is not extended by a non-invoke instruction.
If the added instruction is $P_{m_{j_k}}[i_{j_k}] = \texttt{invoke}\ m'$, we have to consider the various possibilities for $m'$:

- if $m' \notin \Sigma$ and $m' \in M$, $t'$ is left unmodified and $S_{m_{j_k}.i_{j_k}} \supseteq S_{m_{j_k-1}.i_{j_k-1}}$ ensures the result;
- if $m' \in \Sigma$ and $m' \in M$, $m'$ is appended to $t'$ and we know that we have

  $$S_{m_{j_k}.i_{j_k}} \supseteq S_{m_{j_k-1}.i_{j_k-1}} \oplus \Phi(foot_{\mathcal{G}}(\{m'\}));$$

  by Lemma 4,

  $$\Phi(foot_{\mathcal{G}}(\{t'\})) \subseteq S_{m_{j_k-1}.i_{j_k-1}}$$

  entails that

  $$\Phi(foot_{\mathcal{G}}(\{t'\})) \oplus \Phi(foot_{\mathcal{G}}(\{m'\})) \subseteq S_{m_{j_k}.i_{j_k}},$$
- if $m' \notin M$, the fragment $t''$ of the trace that corresponds to the call of $m'$ is such that

  $$\Phi(foot_{\mathcal{G}}(\{t''\})) \subseteq \bigcup_{m'' \leq m'} F_{m''}$$

  since $t''$ is a trace of one such method $m''$; so, by Lemma 4 we get

  $$\Phi(foot_{\mathcal{G}}(\{t'\})) \oplus \Phi(foot_{\mathcal{G}}(\{t''\})) \subseteq S_{m_{j_k-1}.i_{j_k-1}} \oplus \bigcup_{m'' \leq m'} F_{m''} \subseteq S_{m_{j_k}.i_{j_k}}.$$

We thus easily conclude that $\Phi(foot_{\mathcal{G}}(\{t\})) \subseteq U$ and consequently that $F_m \subseteq U$.
Let us prove the converse, $U \subseteq F_m$, by considering the set of definitions $(R_M)$:

$$R_{m'.i} = \bigcup_{p \in paths(ICFG_M, \{m.0\}, \{m'.i\})} \Phi(foot_{\mathcal{G}}(\{m.calls(p)\}))$$

with the fact that

$$F_m = \bigcup_{i | P_m[i] = \texttt{return}} R_{m.i}.$$

We can show that, for any edge $(m'.j', m''.j'')$ in $ICFG_M$, $R_{m''.j''} \supseteq instr_{P_{m'}[j']}(R_{m'.j'})$ by cases over the definition of $instr_b$ in a similar way to previously shown. Since $U$ is the least solution of $(S_M)$, we directly get that $S_{m'.i'} \subseteq R_{m'.i'}$ for all $m'.i' \in ICFG_M$. Therefore

$$U = \bigcup_{i | P_m[i] = \texttt{return}} S_{m.i} \subseteq \bigcup_{i | P_m[i] = \texttt{return}} R_{m.i} = F_m,$$

which concludes the proof. □

This proof uses some property of interest to embed the verification. As we noticed just above, as soon as we have a set of $R_{m'.i'}$ such that, for any edge $(m'.j', m''.j'')$ in $ICFG_M$, $R_{m''.j''} \supseteq instr_{P_{m'}[j']}(R_{m'.j'})$, then the union of footprints at all `return` instructions must contain $F_m$, since it is the least such set. This means that checking for those inclusions will provide a low-complexity technique to ensure that a declared footprint for a method $m$ is safe, *i.e.*, it is an over-approximation of $F_m$. This also means that the footprint implementation against which some method bytecode will be verified can be any such over-approximation: to handle method overloading we will only use the union of all the footprints of methods in one class and its subclasses.

## 5 On-device verification

The computation of footprints uses a fix-point computation, it is thus too heavy for the computation capabilities of target devices such as mobile phones or smart-card. For this reason, we use a proof-carrying code approach [22] analog to Rose [25] developed for Java bytecode type verification.

### 5.1 Encoding of the embedded proof

Each class file is analyzed off-board, either alone or with the methods of its strongly connected component when needed. Then, metadata have to be shipped with the code. For traditional Java environments, they are added to the class file in the form of class file attributes. Java Card platforms (2.x and 3.x Classic) do not accept class files directly but rely on the CAP file structure that is a specific class files bundle. This structure admits "Custom Components" where footprints can be stored and retrieved during the loading process. Whatever the target platform, we need:

- for each method $m$ of the class:
  - the over-approximated footprint $F_m$ of $m$, so that every method $m'$ such that $m' \leq m$ has a footprint included in $F_m$,
  - "proof annotations" that is the list of intermediate footprints computed externally for all $i$ such that $P_m[i]$ is the target of a jump, proof annotations are encoded in the array $proof[i]$ in Algorithm 5.1,
- for each method $m$ that is invoked by methods $m_0, \ldots, m_n$ of the class :
  - "believed footprint" that is the footprint $F_m$ of $m$ that has been used in composition to compute the footprints of the methods $m_0, \ldots, m_n$.

Let us consider a system with a global policy $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$. Footprints are encoded in a binary form, if $S$ contains $n$ elements, $s_0$ is the initial state, $s_F$ is denoted $s_{n-1}$, and then we need $n \times (n-1)$ bits for a footprint and the bit $0 \leq i < n \times (n-1)$ encodes the presence of $(s_{i/n}, s_{i \bmod n})$ in the footprint of $m$. For readability of the rest of this section, we use the array notation and denotes by $F_m[i]$ the $(i+1)$-th bit of $F_m$.

The empty footprint is encoded by $\bot$ (all bits set to zero), and the fully saturated footprint denoted ✠ in the formal model is encoded by the footprint value $\top$ (all bits are set to one).

### 5.2 On-device metadata

To manage methods footprints on-device, we use two repositories: $R$ which maps the verified methods to their footprints, and $R_{tmp}$ the temporary repository which maps methods that are not loaded to their believed footprints.

The global policy $\mathcal{G} = (\Sigma, S, s_0, \zeta, s_F)$ is an automaton on a subset of the methods, thus we need to record it on the embedded system and especially $\Sigma$ and $\zeta$. The global policy $\mathcal{G}$ is encoded by an array of footprints $G$ such that:

$$G[A.C.m] = \begin{cases} \bot & \text{if } m \neq \Sigma \\ \{(s_i, s_j) \mid \zeta(s_i, A.C.m) = s_j \vee \\ \qquad (\zeta(s_i, A'.C'.m') = s_j \wedge A.C.m \leq A'.C'.m')\} & \text{otherwise} \end{cases}$$

The system starts with no application installed, no method has a verified footprint so each method is assigned the encoded empty footprint $\bot$. On the contrary, each method is assigned in the temporary repository the fully saturated footprint: this default value denotes the "worst" possible footprint of method; this assumption is mandatory to be able to reject a method during its verification if it invokes another method for which no footprint is known in $R$ or in $R_{tmp}$. Thus, repositories are initialized such that:

- the repository $R$ of verified methods footprints is empty: $R[m] = \bot$ for all methods $m$;
- the temporary repository $R_{tmp}$ of methods footprints is set to the maximum value: $R_{tmp}[m] = \top$ for all methods $m$, since we will restrict it step by step using intersection.

### 5.3 Verification algorithm

The verification algorithm is given by the Algorithm 5.1. For a policy of $n$ states, a footprint is a vector of $n \times (n-1)$ bits. Then the basic operations are encoded in the following way:

- nonvalid($F_m$) is $(F_m \& mask == mask)$ with $mask$ the vector that encodes $\{(s_0, s_{n-1})\}$, *i.e.*, all bits equal to 0 except the bit $n-1$,
- $F_1 \subseteq F_2$ is $F_1 \mid F_2 == F_2$.

```
 1: for all believed footprint F_{A'.C'.m'} in the class file do
 2:     if R[A'.C'.m'] ⊈ F_{A'.C'.m'} then return FAIL
 3:     R_tmp[A'.C'.m'] = R_tmp[A'.C'.m'] & F_{A'.C'.m'}
 4: for all footprint F_{A.C.m} in the class file do
 5:     if nonvalid(F_{A.C.m}) or F_{A.C.m} ⊈ R_tmp[A.C.m] then return FAIL
 6:     if A.C.m ≤ A'.C'.m' and F_{A.C.m} ⊈ F_{A'.C'.m} then return FAIL
 7: read proof annotations in array proof
 8: for all method A.C.m defined in class C do
 9:     F_tmp = G[A.C.m]
10:     for all bytecode i from 0 to end do
11:         if ∃proof[i] then
12:             if F_tmp ⊈ proof[i] then return FAIL
13:             F_tmp = proof[i]
14:         if P_{A.C.m}[i] = invoke A'.C'.m' then
15:             if R[A'.C'.m'] ≠ ⊥ then
16:                 F_tmp = compose(F_tmp, R[A'.C'.m'])
17:             else
18:                 F_tmp = compose(F_tmp, R_tmp[A'.C'.m'])
19:         else if P_{A.C.m}[i] = return and F_tmp ⊈ F_{A.C.m} then
20:             return FAIL
21:         else if P_{A.C.m}[i] ∈ branching bytecodes to address a then
22:             if ∄proof[a] or F_tmp ⊈ proof[a] then return FAIL
23:             if P_{A.C.m}[i] ∈ branches systematically to a ≠ i + 1 then F_tmp = ⊥
24:         if nonvalid(F_tmp) then return FAIL
25: drop proof and add all F_{A.C.m} to R
26: drop all R_tmp[A.C.m]
27: return SUCCESS
```

**Algorithm 5.1:** Loading of a class $C$ of application $A$.

From lines 2 to 3, we check the believed footprints: if a verified footprint for the same method already exists on the device, the shipped one must conform to it, then the believed

footprint in $R_{tmp}$ for $A'.C'.m'$ ($\top$ by default) is restricted to its common parts with the believed footprint $F_{A'.C'.m'}$ coming with the currently verified application; since this restriction can only remove factors contributing to invalid traces, it cannot cause methods already loaded and successfully verified with the previous footprint to be rejected if they were re-verified with the restricted footprint. Then, for each believed footprint, we check line 5 if it is valid, if it conforms to the believed ones of the applications already loaded on the system, and if it is compliant with the class hierarchy line 6.

Then, the last part of the algorithm verifies the proof of the footprint of the method without any fix-point computation. For each branching bytecode, we just have to verify that the computed footprint is lower than the proof annotation. To avoid cycles in the verification algorithm, for each bytecode that has a proof annotation, we use it as current state instead of the computed one as seen line 13, after verification that the current state is included in the proof annotation line 12. Only invocation bytecodes have an impact on the footprint: we compose (see Algorithm 5.2) the current footprint with the footprint of the invoked method if it is already verified line 16, with the believed ones otherwise line 18. For the return bytecode, we only have to verify that the current state is compliant with the footprint announced for the method line 19.

Lastly, for branching bytecodes, we have to check that the target bytecode has been annotated and that the current state is included in the proof annotation of the target line 22, and if the bytecode always branches (`goto`, `return`, *etc.*), we have to reset the current state line 23, before analyzing the next bytecode.

Algorithm 5.2 implements the composition in a different way from the formal definition. The composition of $F_1$ and $F_2$ computed here is only the union of:

- $\{(s_i, s_j) \mid \exists k, (s_i, s_k) \in F_1, (s_k, s_j) \in F_2\}$, implemented by lines 2 to 5,
- $\{(s_0, s_i) \in F_2\}$, line 7,
- $\{(s_i, s_F) \in F_1\}$, line 8.

Instead of saturating the composition as soon as $(s_0, s_F)$ is found in it, since the actual implementation must directly reject the code, we simply test after this computation whether $(s_0, s_F)$ was added and fail if this is the case.

1: Let $F_{res}$ be initialized to 0
2: **for all** $0 \leq i < n - 1$ **do**
3: **for all** $0 \leq j < n$ such that $F_1[i \times n + j] == 1$ **do**
4: **for all** $0 \leq k < n$ such that $F_2[j \times n + k] == 1$ **do**
5: $F_{res}[i \times n + k] = 1$
6: **for all** $0 \leq i < n$ **do**
7: $F_{res}[i] = F_{res}[i] | F_2[i]$
8: $F_{res}[i \times n + n - 1] = F_{res}[i \times n + n - 1] | F_1[i \times n + n - 1]$

**Algorithm 5.2:** Composition of footprint: compose $(F_1, F_2)$.

### 5.4 Extension of the verification to removal of applications

We have proposed a model that allows an issuer to define a global policy of a system, and an incremental verification designed to avoid re-verification of already loaded bytecode. In order

to keep this last property while dealing with applications removal, we have to keep additional metadata on-device.

When an application $A$ is successfully verified and installed, all the believed footprints of $A$'s methods are removed from the temporary repository $R_{tmp}$, and the verified footprints of $A$'s methods are stored in $R$. Moreover, if $A$ invokes some shared methods of other applications not yet installed, then the believed footprints of these methods have been updated to intersect with the believed footprints brought by $A$. In order to remove $A$, we have to be able to restore the temporary repository. To achieve this reset, it is mandatory to keep all the individual believed footprints of shared methods, and not only an aggregation of these footprints, brought by applications until there are uninstalled.

Concretely, a new repository $F_{restore}$ is defined to gather the collection of believed footprints coming with each application. We choose to still maintain $R_{tmp}$ in a incremental way (line 3 of Algorithm 5.1) to avoid multiple comparisons (line 5 of Algorithm 5.1) with all believed footprints now stored in $F_{restore}$. However, $R_{tmp}$ can be removed to reduce memory requirements if necessary, which will conversely increase the number of required comparisons. Indeed, in case $R_{tmp}$ is removed, each expression of the form $F_{A.C.m} \not\subseteq R_{tmp}[A.C.m]$ has to be replaced by its equivalent: there exist an installed application $A'$ such that $F_{A.C.m} \not\subseteq F_{restore}[A.C.m][A']$. Composition is not done with verified footprints anymore but with the believed ones brought on by the application itself. Thus, we apply the following modifications in Algorithm 5.1:

$$\begin{aligned} \text{lines } 15 - 18: &\longrightarrow F_{tmp} = \mathit{compose}(F_{tmp}, F_{A'.C'.m'}) \\ \text{line } 26: &\longrightarrow \text{store each } F_{A'.C'.m'} \text{ in } F_{restore}[A'.C'.m'][A] \end{aligned}$$

Upon removal of an application $A$, the believed footprints of the external methods invoked by $A$ must be restored. This last operation requires to recompute the believed footprints according to the collection of believed footprints of all external methods brought by applications remaining installed, as depicted in Algorithm 5.3. Finally, the removal of an application $A$ must reset the verified footprint of each method of $A$ to its default value $\bot$ in $R$ (line 5 of Algorithm 5.3).

1: **for all** $F_{restore}[A.C.m]$ **do**
2: $\quad R_{tmp}[A.C.m] = \top$
3: $\quad$ **for all** $A' \in \mathcal{A}$ **do**
4: $\quad\quad R_{tmp}[A.C.m] = R_{tmp}[A.C.m] \& F_{restore}[A.C.m][A']$
5: **for all** $A.C.m$ **do**
6: $\quad R[A.C.m] = \bot$

**Algorithm 5.3:** Rollback after the removal of an application.

## 6 Conclusion

In this paper we have proposed a powerful technique to enforce control flow policies on Java bytecode in open and constrained systems. Our approach is purely static so it does not require any execution monitoring, which is a better approach for strongly constrained devices such as smart cards. Moreover, the incremental and compositional verification scheme of our approach permits to efficiently deal with post-issuance (un)installation of applications without the need to re-verify already loaded code. As we use a proof-carrying code approach, no code-signing mechanism is required for the device to be protected against code originating

from an untrusted origin or transmitted through an unsecure communication channel, which is crucial for open devices. The main issue we still have to face is the monolithic security policy stored on-device that impacts security policy updates. Further works will focus on an efficient way to update the security policy on-device.

## References


1. Alpern, B., and Schneider, F. B. Recognizing safety and liveness. *Distributed Computing 2*, 3 (1987), 117–126.
2. Alpern, B., and Schneider, F. B. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages and Systems (TOPLAS) 11*, 1 (January 1989), 147–167.
3. Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., and Stark, I. Mobile resource guarantees for smart devices. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop (CASSIS'04)* (Marseille, France, March 2005), G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., vol. 3362 of *Lecture Notes in Computer Science*, Springer, pp. 1–26.
4. Bauer, L., Ligatti, J., and Walker, D. Composing security policies with Polymer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)* (Chicago, IL, USA, June 2005), V. Sarkar and M. W. Hall, Eds., ACM, pp. 305–314.
5. Bielova, N., Dragoni, N., Massacci, F., Naliuka, K., and Siahaan, I. Matching in security-by-contract for mobile code. *Journal of Logic and Algebraic Programming 78*, 5 (May–June 2009), 340–358.
6. Bielova, N., and Massacci, F. Do you really mean what you actually enforced? In *5th International Workshop on Formal Aspects in Security and Trust (FAST'08)* (Malaga, Spain, October 2008), P. Degano, J. D. Guttman, and F. Martinelli, Eds., vol. 5491 of *Lecture Notes in Computer Science*, Springer, pp. 287–301.
7. Brown, A., and Ryan, M. Synthesising monitors from high-level policies for the safe execution of untrusted software. In *4th International Conference on Information Security Practice and Experience (ISPEC'08)* (Sydney, Australia, April 2008), L. Chen, Y. Mu, and W. Susilo, Eds., vol. 4991 of *Lecture Notes in Computer Science*, Springer, pp. 233–247.
8. Colcombet, T., and Fradet, P. Enforcing trace properties by program transformation. In Wegman and Reps [31], pp. 54–66.
9. Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., and McDaniel, P. D. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)* (Vancouver, BC, Canada, October 2010), USENIX Association.
10. Erlingsson, Ú., and Schneider, F. B. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy (S&P'00)* (Oakland, California, USA, May 2000), IEEE Computer Society, pp. 246–255.
11. Fong, P. W. L. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy (S&P'04)* (Berkeley, California, USA, May 2004), IEEE Computer Society, pp. 43–55.
12. Gurov, D., Huisman, M., and Sprenger, C. Compositional verification of sequential programs with procedures. *Information and Computation 206*, 7 (2008), 840–868.
13. Huisman, M., and Tamalet, A. A formal connection between security automata and JML annotations. In *12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)* (York, UK, March 2009), M. Chechik and M. Wirsing, Eds., vol. 5503 of *Lecture Notes in Computer Science*, Springer, pp. 340–354.
14. Jensen, T. P., Le Métayer, D., and Thorn, T. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy (S&P'99)* (Oakland, California, USA, May 1999), IEEE Computer Society, pp. 89–103.
15. Klein, G., and Nipkow, T. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience 13*, 13 (2001), 1133–1151.
16. Ligatti, J., Bauer, L., and Walker, D. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security 4*, 1–2 (February 2005), 2–16.
17. Ligatti, J., Bauer, L., and Walker, D. Enforcing non-safety security policies with program monitors. In *10th European Symposium on Research in Computer Security (ESORICS'05)* (Milan, Italy, September 2005), S. De Capitani di Vimercati, P. F. Syverson, and D. Gollmann, Eds., vol. 3679 of *Lecture Notes in Computer Science*, Springer, pp. 355–373.

18. Massacci, F., and Siahaan, I. Simulating midlet's security claims with automata modulo theory. In *Workshop on Programming Languages and Analysis for Security (PLAS'08)* (Tucson, AZ, USA, June 2008), Ú. Erlingsson and M. Pistoia, Eds., ACM, pp. 1–9.
19. Mizuno, M., and Schmidt, D. A. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing 4*, 6A (November 1992), 727–754.
20. Morrisett, G., Walker, D., Crary, K., and Glew, N. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS) 21*, 3 (May 1999), 527–568.
21. Myers, A. C. JFlow: Practical mostly-static information flow control. In *27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)* (San Antonio, Texas, USA, January 1999), A. Appel and A. Aiken, Eds., ACM, pp. 228–241.
22. Necula, G. C. Proof-carrying code. In *24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)* (Paris, France, January 1997), P. Lee, F. Henglein, and N. D. Jones, Eds., ACM, pp. 106–119.
23. Ongtang, M., McLaughlin, S. E., Enck, W., and McDaniel, P. D. Semantically rich application-centric security in Android. In *25th Annual Computer Security Applications Conference (ACSAC'09)* (Honolulu, Hawaii, December 2009), IEEE Computer Society, pp. 340–349.
24. Pottier, F., Skalka, C., and Smith, S. F. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems (TOPLAS) 27*, 2 (March 2005), 344–382.
25. Rose, E. Lightweight bytecode verification. *Journal of Automated Reasoning 31*, 3–4 (January 2003), 303–334.
26. Schneider, F. B. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC) 3*, 1 (February 2000), 30–50.
27. Sekar, R., Venkatakrishnan, V. N., Basu, S., Bhatkar, S., and DuVarney, D. C. Model-carrying code: a practical approach for safe execution of untrusted applications. In *19th ACM Symposium on Operating Systems Principles (SOSP'03)* (Bolton Landing, NY, USA, October 2003), M. L. Scott and L. L. Peterson, Eds., ACM, pp. 15–28.
28. Talhi, C., Tawbi, N., and Debbabi, M. Execution monitoring enforcement for limited-memory systems. In *International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services (PST'06)* (Markham, Ontario, Canada, October 2006), G. Sprague, B. Schell, and W. Fond, Eds., vol. 380, ACM, pp. 38:1–38:12.
29. Vanoverberghe, D., and Piessens, F. Supporting security monitor-aware development. In *3rd International Workshop on Software Engineering for Secure Systems (SESS '07)* (2007), IEEE Computer Society, pp. 2–6.
30. Walker, D. A type system for expressive security policies. In Wegman and Reps [31], pp. 254–267.
31. Wegman, M., and Reps, T., Eds. *27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)* (Boston, MA, USA, January 2000), ACM.
32. Winwood, S., Klein, G., and Chakravarty, M. M. T. On the automated synthesis of proof-carrying temporal reference monitors. In *16th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'06)* (Venice, Italy, July 2006), G. Puebla, Ed., vol. 4407 of *Lecture Notes in Computer Science*, Springer, pp. 111–126.

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399